The Motor Industry Software Reliability Association

MISRA c/o Electrical Group, MIRA, Watling Street, Nuneaton, Warwickshire, CV10 0TU, UK.
Telephone: (024) 7635 5290. Fax: (024) 7635 5070. E-mail: misra@mira.co.uk Internet: http://www.misra.org.uk

Report 5

Software Metrics

February 1995

PDF version 1.0, January 2001

Acknowledgements

The following were contributors to this report:

Frank O'Neill	Lucas Electronics
Nick Bennett	Lucas Electronics
David Newman	Ford Motor Company Ltd
Ian Kendall	Jaguar Cars Ltd
Roger Rivett	Rover Group Ltd
Paul Edwards	Rover Group Ltd
Chris Marshall	AB Automotive Electronics Ltd
Paul Manford	AB Automotive Electronics Ltd
Peter Jesty	University of Leeds
Keith Hobley	University of Leeds
Neil Andrewartha	Rolls Royce and Associates Ltd
Vivien Hamilton	Rolls Royce and Associates Ltd
John Fox	The Centre for Software Engineering Ltd
Brian Whitfield	The Centre for Software Engineering Ltd
Anthony Wood	The Centre for Software Engineering Ltd

Summary

This document identifies a number of software attributes and metrics which may be used to provide a measure of those attributes and hence of the quality of software.

At present the majority of automotive software is developed in assembler for reasons of speed and size. However, there has been some movement towards the use of high level languages. The software orientated metrics presented in this document have been selected due to their applicability (in most cases) to both low level and high level languages.

The report also deals with collection and interpretation of metric results, with an emphasis on interpretation of results and how data can be fed back into the development process in order to improve it. Some of the tools available which support such collections and interpretation are identified.

The set of metrics detailed in the document are not an exhaustive set. In this context it is intended that the document should be a reference document to aid in the determination of appropriate metrics to be used on software projects. Several further reference documents are identified which can provide more detailed information and guidance on this topic.

Recommendations

This report shows that it is only possible to effectively implement metric analysis within a project if sufficient management techniques are used and the staff employed have a detailed understanding of the objectives of the metric plan. Metrics should be seen as a support tool for projects, not as a burden.

The success or failure of using metrics depends upon how well the analysis was originally planned and how effectively it is implemented. Procedures and guidelines should be used to force a structured approach to the analysis so that full documentation and traceability is produced.

The methods of presenting collected information are extremely important for ease of interpretation. Once metrics have been collected and analysed correctly they provide a firm basis for future project planning. The effectiveness of feedback of results should be closely monitored to ensure that lessons are being learned, and improvement actions taken where appropriate.

Software process metrics

Software process metrics are measures which provide information about the performance of the development process itself. There is a dual purpose to the collection of such data; firstly they provide an indicator as to the ultimate quality of the software being produced, and secondly they can assist the organisation to improve its development process by highlighting areas of inefficiency or error-prone areas of the process.

It is recommended that at least core process metrics are collected and analysed (in order to be useful, the analysis must be performed in a timely manner in order to facilitate effective feedback). A number of reference books are available which give a more detailed description of process metrics and how to collect the information, for example Fenton [1], Grady and Caswell [2], Metric Users Handbook [3] and IEEE Std 982.1-1988 [4]. The basic metrics recommended are:

- estimated duration of each task
- actual effort expended on tasks
- number of defects detected

Other metrics may be found useful, but this will depend on the organisation. There are no "correct" metrics. Initiating a metrics programme can be problematic, so for an organisation not currently collecting software metrics:-

1. Identify those metrics which are felt to be both meaningful and useful within the organisation.
2. Do not be too ambitious initially - start with a number of simple metrics (such as

those listed above).
3. Do not concentrate on a single metric - this can distort the collected data.
4. Analyse data in a timely manner so that feedback can be effective.

When measuring data on defects, it can be extremely effective also to record the source of the defect (lifecycle phase) and the phase the defect was detected. Using this data, it is possible to assess the effectiveness of the development process at different phases. If coupled with data from item reviews, problem areas of the development process can be quickly identified and corrective action taken.

Other aspects which should be borne in mind are:-

1. It is not advisable to allow metrics to be used to measure performance of individuals.
2. Set clear goals and involve staff in the choice of metrics.
3. Provide regular feedback to the development team and discuss problems and performance issues.
4. Data collection methods should be as simple as possible, preferably automated, to minimise the effort involved in collection.
5. Data should be presented graphically if possible for ease of interpretation, particularly since trends will often be more relevant than absolute values.

Product metrics

Product metrics are a measure of some attribute of an output of the software development process (e.g. code). The attributes which can be measured include:

- Complexity
- Maintainability
- Modularity
- Reliability
- Structuredness
- Testability

The metrics can then be used to assess the quality of the output. For example, a highly complex code item with poor structure is more likely to contain errors than a simple item with good structure. Product metrics therefore can be used in conjunction with process metrics to manage the development process.

Contents

Page

1. Introduction

The overall requirement of a software system is that the system performs the required functions with the required accuracy within the required response times. In order to achieve this requirement, the software system should possess a high quality of design and the developers should have used what is considered as good software engineering practice.

In many respects high quality design and good programming practice are an exercise in compromise between a number of conflicting requirements. Often there is no clear means of determining whether they have been achieved if the development process is not defined in detail. The approach which is generally taken for achieving this compromise is through the use of suitable development standards and quality assurance procedures [5], and a rigorous verification and validation process.

The overall quality of a software system is impossible to quantify with confidence. However, a measurement of the overall quality of a software system can be obtained by measurement of those attributes of the software which individually contribute to its quality. This kind of approach is proposed by Boehm [6] and McCall [7].

In the context of software engineering, the term 'metrics' is usually used to describe virtually anything which is in any way measurable. Whereas in other engineering disciplines, the measurement of aspects of both the process and product is commonplace, this is not so in software engineering where metrication remains the domain of a relatively small number of specialists.

While a great deal of attention has been given to improving the software engineering process in terms of new methods, tools and techniques, the quantification of improvement through the measurement of process attributes has been largely neglected. Without such measurement it is clearly difficult to assess the impact of process changes.

Clearly some things are easily measurable, others less so. However, to desist from measuring something simply because it is difficult does little to advance the state of understanding of the attributes (for example software reliability and quality). The following statements are still unfortunately true of a large number of software engineering projects:

i) We still fail to set measurable targets when developing software products. For example, statements such as 'user-friendly', 'reliable', and 'maintainable' are made without thought to what these mean in measurable terms.

ii) We fail to measure the various components which make up the real costs of software projects. For example, we do not know how much time was really spent on design compared with testing.

iii) We do not attempt to quantify the quality (in any sense) of the products we produce, e.g. the likelihood of failure of the system over a given period.

iv) We still rely on purely anecdotal evidence to convince us to try yet another revolutionary new development technology or tool.

Where measurement is performed, it appears to be done in a largely ad hoc, uncontrolled and therefore unrepeatable manner. Without standardised measurements and measuring techniques it is impossible to make comparisons within the same company, let alone across industry as a whole.

1.1 Purpose of Software Measurement

There seem to be two main purposes for software measurement - assessment and prediction. Measurements for assessment help to keep track of a particular software project. Other measurements, however, are used for predicting important characteristics of projects. It is important not to exaggerate the potential of such predictions. In particular, it must be recognised that the application of certain models or tools will not yield accurate predictions without accurate measurement for calibration purposes. In other words, you cannot predict what you cannot measure.

1.2 Scope of Software Metrics

The term 'Software Metrics' is applied to a diverse range of activities including:

- Cost and effort estimation
- Productivity measures and models
- Quality models and measures
- Reliability models
- Performance evaluation models
- Structural and complexity metrics

The following sections summarise the current position of the above with regard to techniques and approaches to measurement.

1.2.1 Cost and Effort Estimation

Due mainly to the need for managers to predict project costs at an early stage in the software life-cycle, numerous models for software cost and effort estimation have been proposed and used e.g. COCOMO [8]. The basic COCOMO model is intended to give an order of magnitude estimation of software costs. It uses only the estimated size of the software project in terms of lines of code and type of software being developed.

Versions of the COCOMO estimation formula exist for three classes of software project:

i) Organic Mode Projects. These are projects where relatively small teams are working in a familiar environment developing applications with which they are familiar. In short, communication overhead is low, and team members know what they are doing and can quickly get on with the job.

ii) Semi-detached Mode Projects. These projects represent an intermediate stage between organic mode projects and embedded-mode projects, described below. The project team may be made up of experienced and inexperienced staff. Team members have limited experience of related systems and may be unfamiliar with some (but not all) aspects of the system being developed.

iii) Embedded Mode Projects. The principal characteristic of embedded mode projects is that they must operate within tight constraints. The software system is part of a strongly coupled complex of software, hardware, software, regulations and operational procedures. Because of the diverse nature of embedded mode projects, it is unusual for project team members to have a great deal of experience in the particular application which is being developed.

1.2.2 Productivity Measures and Models

The need of management to control the cost of software projects has also resulted in measures and models for assessing the productivity of engineers. Sadly, however, simplistic measures such as lines of code produced per day fail to take account of other factors such as experience, problem complexity and quality. They can however be useful for providing relative comparisons between systems which have some common attributes (e.g. rough size and complexity).

1.2.3 Quality Models and Measures

The more considered attempts to estimate cost and measure productivity concluded that no accurate, meaningful models and measures would be possible without due consideration of the quality of the software produced. Work in this field has led to the concept of high level quality factors of software products, for example reliability and useability, which we would ideally like to measure. Such factors are governed by lower level criteria such as modularity, data commonality etc. Criteria are, in principle, easier to measure than factors. More advanced quality models therefore describe all pertinent relationships between factors and their dependent criteria.

1.2.4 Reliability Models

Underlying the general work on software quality models is the detailed work concentrating on the measurement and prediction of specific software product attributes. The work on measuring and predicting software reliability can be seen as a rigorous and fairly successful example of a detailed study of one particularly important product quality attribute.

1.2.5 Performance Evaluation Models

Performance Evaluation Models are concerned with measurement of a specific software product attribute. Work in this area includes externally observable system performance aspects like response times and completion rates. Also, work on computational and algorithmic complexity, i.e. the internal workings of a system, is concerned with the inherent complexity of problems measured in terms of efficiency of an optimal solution.

1.2.6 Structural and Complexity Metrics

The work on structural complexity metrics is best seen as underlying the work on measuring specific quality attributes. Desirable quality attributes like reliability and maintainability cannot be measured until some operational version of the code is available. However, we wish to be able to predict which parts of the software system are likely to be less reliable or require more maintenance than others. Hence the idea is that measurements of the structural attributes of representations of the software which are available in advance of (or without the need for) execution. This approach assumes that from this complexity data, empirically predictive theories to support quality assurance, quality control and quality prediction can be developed. Examples of this approach are the works of Halstead [9] and McCabe [10].

1.3 Types of Software Metrics

It is beneficial to classify metrics according to their usage. IEEE928.1 [4] identifies two classes:

i) process - activities performed in the production of the software.

ii) product - an output of the process, for example the software or its documentation.

In addition to these two classes of metrics Fenton [1] introduces a further class; Resources. This class of metrics is not considered in this document as the metrics may be included in the process class.

In explaining some of the activities required for the assessment of software reliability [11] identifies product and process properties which can be interpreted as metrics.

Many of the metrics presented in this document for assessing the achievement of the software attributes are detailed in [4].

2. Process Metrics

Software process metrics are measures which provide information about the performance of the development process itself. There is a dual purpose to the collection of such data; firstly they provide an indicator as to the ultimate quality of the software being produced, and secondly they can assist the organisation to improve its development process by highlighting areas of inefficiency or error-prone areas of the process.

A number of reference books are available which give a more detailed description of process metrics and how to collect the information, for example Fenton [1], Grady and Caswell [2], Metric Users Handbook [3] and IEEE Std 982.1-1988 [4]. A number of basic process metrics are described below.

2.1 Development Time

The time spent on developing a specific item (such as a specification), a specific lifecycle phase, or the total project. Such time will also include reworking time. As well as simply recording time, the cost of the development can be recorded.

This information may then be used to assess the accuracy of estimates made prior to work commencing, which in turn should lead to more effective estimation of future projects. This is particularly relevant for safety critical projects where the effort expended is generally related to the criticality level of the software.

2.2 Number of Problem Reports

This metric records the number of problems reported in a period, lifecycle phase, or project experienced, and considers:

i) source of problem;

ii) nature of the problem;

iii) where observed in lifecycle; and

iv) the number of components affected.

By analysing data relating to software problem reports, it is possible to determine the source of the problem (e.g. customer requirement, design, code, test etc) and to correlate this against the point at which the problem was identified. If, for example, a significant number of design problems are being identified at system test, then the review process which permitted such errors to be perpetuated through the lifecycle should be reviewed.

2.3 Effort Due to Non Conformity

The effort spent in reworking the software due to improvement to quality or as a result of errors found. The effort can be expressed as hours or cost.

In addition, it is often useful to gather data relating to the effectiveness of the review process, i.e. how many iterations were required before a particular component (report, design, code item etc) was accepted. This might highlight problems with team organisation for example, which might then be addressed through additional training.

2.4 Number of Changes

This metric records the number of changes made in a period, lifecycle phase or project. As with problem reports, this data can be used to assess the source of changes. For example, a large number of changes being introduced by the customer as requirement changes could indicate that the system is not yet mature.

2.5 Slippage

This metric indicates a deviation between the initial prediction and reality. The metric may be based on time (lateness) or effort (hours or cost).

3. Product Metrics

The attributes of the software product (i.e. the source code) can be assessed qualitatively by reviews and/or quantitatively by expressing attributes as measurable quantities, that is by metrics. This section identifies methods and metrics which can be used to measure software attributes. Appendix A gives detailed definitions of a number of the metrics suggested in this section.

The attributes considered in this section are:

i) Complexity;

ii) Maintainability;

iii) Modularity;

iv) Reliability;

v) Structuredness;

vi) Testability;

vii) Understandability; and

viii) Maturity.

3.1 Complexity

Complexity is a major cause of unreliability in software [10]. The complexity of an object is some measure of the mental effort required to understand and create that object [12].

In a software system there are four identifiable types of complexity which encompass both software components and the software system.
These types of complexity are:

i) text complexity;
ii) component complexity;
iii) system complexity; and
iv) functional complexity.

Myers [12] and Shooman [13] considered only types i), ii) and iii) since type iv) (functional complexity) has no recognised metric associated with it.

Each of the types of complexity i), ii) and iii) are examined below. For each type a list of

metrics is given which could be used to obtain an overall view as to whether the complexity has achieved an acceptable level.

3.1.1 Text Complexity

The Text Complexity of a software component is closely linked to both the size of the component and to the number of its operators and operands.

Some metrics which can be used to analyse the text complexity are:

i) Number of Statements;
ii) Number of Distinct Operands;
iii) Number of Distinct Operators;
iv) Number of Operand Occurrences;
v) Number of Operator Occurrences;
vi) Vocabulary Size;
vii) Component Length; and
viii) Average Statement Size.

3.1.2 Component Complexity

The Component Complexity is a measure of the inter-relationships between the statements in a software component [12]. The metrics which could be used to analyse the component complexity are:

i) Cyclomatic Number;
ii) Number of Decision Statements; and
iv) Number of Structuring Levels.

There also exists a metric which identifies whether the component is a potential stress point in the system [14]. This is the Components Stress Complexity and is related to the Cyclomatic Number.

3.1.3 System Complexity

The System Complexity is a measure of the complexity of relationships between the software components, [12]. The metrics which could be used to analyse the system complexity are:

i) Number of Components;
ii) Number of Calling Levels;
iii) Number of Calling Paths;
iv) Hierarchical Complexity; and
v) Structural Complexity.

3.2 Maintainability

Maintenance of a software system involves changing installed software to either correct an error (corrective), prevent a failure (preventative) or to add an enhancement (adaptive). It is very important that maintainability, that is the ease with which a change can be accomplished in a controlled manner, is optimised. This optimisation can be facilitated by the software being simple, understandable, structured, modular, testable and adequately documented and commented.

The maintainability requirements encompass the following other software attributes:

i) Complexity;
ii Modularity;
iii) Structuredness;
iv) Testability; and
v) Understandability;

Whether a system is maintainable or not is a qualitative issue which should be determined on a system by system basis and involve the analysis of the software attributes which impact the system's maintainability.

3.3 Modularity

Modularity is the degree to which a software system is decomposed into a set of independent components. To achieve Modularity it is necessary to have a single function per component (Component Strength) and to minimise the data relationships among components by using formal parameter passing (Component Coupling).

3.3.1 Component Strength

Component strength is an indication of the internal relationships of a component. It is a measure of the extent to which a component performs a single function. A strong component is a component which only performs a single identified function, for example, a function which clears the screen of a terminal. The strength of a component is a qualitative decision made after the function/functions performed by the component have been identified.

An indication of a component's strength can be obtained by examining the number of components which call a component (Fan-in) and the number of components called by the component (Fan-out) [14]. For instance, a high fan-in and fan-out usually reveals a lack of functionality in that the component may perform more than one function due to its large number of connections. If a high fan-in or high fan-out exists then the component may lack refinement and may be better split into two or more components. It should be noted that library routines, etc. have a high Fan-In but should always occur at the lowest calling level.

3.3.2 Component Coupling

Component Coupling is a measure of the relationships between components. An analysis of the components is necessary in order to determine the level of coupling between them.

This document will use the seven levels of coupling defined by Myers [12]. These are:

i) No direct coupling (Best) - Two components do not relate to each other but each independently relates to the calling program;

ii) Data Coupling - Two components pass items of data through their input-output interface. Only the required data is passed and not whole data structures;

iii) Stamp Coupling - Two components pass a data structure through their input-output interface;

iv) Control Coupling - When one component passes switch, flag or component name information to another. Languages which have conditions which can be raised can also be control coupled;

v) Internal Coupling - This is possible in a few languages where the variable coupling can be controlled and limited to those formally declared as external;

vi) Common Coupling - When components reference the same data, either via global or common declarations; and

vii) Content Coupling (Worst) - If a component can branch to another component rather than calling that component.

For all software systems it is recommended, by Myers [12], that the only levels of Component Coupling allowed are i), ii) and iii), as the remaining levels result in a system which is more complex and less understandable.

3.4 Reliability

Software is inherently different to hardware as it can only suffer from systematic failures unlike hardware which can suffer from random failures. Due to the nature of software it is impossible to determine an overall quantitative reliability value for its systematic failure. At present there are two possible methods for improving, or obtaining an indication of, the reliability of software.

Software reliability can be improved by using design strategies such as defensive programming, diversity or redundancy of the software. Defensive programming allows the software to take specific actions if a failure occurs. Diversity of the software removes some of the risk of common cause failures. Software redundancy allows a version of the software

to continue operation if another version of the software becomes corrupt.

The reliability of software can be estimated by using reliability estimation models. However these need large amounts of empirical data in order to yield useful results and are not always practicable.

A wide variety of these models have been used and documented by a variety of companies on previous projects. The underlying principle for deriving reliability growth is to compare the current time between failures with the previous time between failures. An increase between the current time and the previous time between failures implies a reliability growth. To evaluate the reliability growth a variety of values must be known or estimated, for example:

i) a chronological list of all previous inter-failure times;

ii) an estimate of the number of initial faults;

iii) debugging time in months from start of integration;

iv) total number of failures in maintenance life;

v) the average contribution to the failure rate of each fault; and

vi) modes of operation.

3.5 Structuredness

Structuredness is the adherence to the structured programming philosophy [12]. It aims to provide a well defined, clear and simple approach to software design, which should be easier to understand, evaluate and modify. Structured programming requires the use of single entry and single exit control structures. The classic technique to implement these procedures is to restrict the control structures used to a set of three general statements, a block sequence, an IF-THEN-ELSE and LOOP statements, such as FOR, WHILE and REPEAT. This classical set is generally extended to include the CASE statement which is in essence a set of nested IF-THEN-ELSE statements.

The GOTO statement is not allowed as the indiscriminate use of the GOTO statement leads to unduly complicated control structures [12]. It has been shown that GOTO statements can be eliminated and replaced by a single decision structure and a single looping structure [15].

A useful technique in deciding whether a component is structured is by the internal reduction of a control graph. A structured control graph should be reducible to a sequence of nodes. Otherwise reduction will show elements of the control graph which do not comply with the rules of structured programming.

The principle of control graph reduction is to simplify the most deeply nested structure control subgraphs into a single reduced node accompanied by the entry and exit nodes. The Essential Cyclomatic Complexity metric gives an indication of the structuredness of a software component. If the result of this metric is 1, then the component is structured. However, if a component possesses multiple entry or exit points then the component cannot be reduced fully. That is it cannot achieve an Essential Cyclomatic Complexity result of 1. The use of multiple entry and exit points breaks the most fundamental rule of structured programming.

To achieve structuredness in software a suitable programming language should be used. Some programming languages with their suitability for structured programming are listed below. It should be noted that some of the languages suitable for structured programming also contain features which are not recommended. The use of such features should be prohibited by the use of Coding Standards.

i) Good suitability, most structures available for example, ADA, Algol, C, Fortran 77, Pascal, PLM, PL/I;
ii) Poor suitability, some structures available for example, Basic, Cobol, Fortran IV; and
iii) Not suited, lacking important structures for example, Assembler Languages, APL.

Metrics which could be used to analyse the structuredness of the software components are:

i) Cyclomatic Number;
ii) Essential Cyclomatic Complexity;
iii) Number of Entry Points;
iv) Number of Exit Points;
v) Number of Structuring Levels;
vi) Number of Unconditional Jumps; and
vii) Number of Execution Paths.

3.6 Testability

Testability requires that both the software system and each software component are designed to facilitate testing in a controlled manner. The design of the system should be such that functions can be fully tested to ensure that they meet their requirements. Testable systems are generally well structured, modular, simple and consistent.

The software system is required to be tested both at component level (unit testing) and at system level (integration testing). Measures for how well software can be tested can be obtained prior to performing the testing (static metrics).

Measures for how well the testing has been performed, that is, the adequacy of the test cases and test coverage can be obtained (dynamic metrics).

One should note that there are other forms of static and dynamic analysis.

3.6.1 Static Metrics relating to Testability

The two lists below contain metrics which can give an indication of how well the software components and the software system can be tested. These indications can be obtained prior to testing.

The following metrics can be applied on a component basis:

i) Cyclomatic Number;
ii) Number of Distinct Operands;
iii) Number of Unconditional Jumps;
iv) Number of Decision Statements; and
v) Number of Execution Paths.

The following metrics can be applied on a system basis:

i) Number of Calling Paths; and
ii) Number of Components.

3.6.2 Dynamic Metrics Relating to Testability

The list below contains metrics which can be used to demonstrate the level of test coverage achieved on either a component or system basis. The metrics can be used to obtain a level of test coverage:

i) Instruction Block (IB) Coverage;
ii) Decision to Decision Path (DDP) Coverage;
iii) Linear Code Sequence and Jump (LCSAJ) Coverage; and
iv) Procedure to Procedure Path (PPP) Coverage.

3.7 Understandability

The software system, its design and documentation should be readily understood by those required to verify, validate and maintain it. Static metrics can be used to give an indication of understandability of the software. The following metrics could be applied:

i) Number of Statements;
ii) Comment Frequency;
iii) Number of Distinct Operands;
iv) Number of Distinct Operators;
v) Vocabulary Size;
vi) Average Statement Size;
vii) Component Length; and

viii) Number of Components.

A qualitative indication of the understandability of the user documentation should be obtained in order to give an indication of the degree of user comprehension and accuracy. This could be performed as part of a Technical Audit.

3.8 Maturity

Maturity applies to existing software components used in the software system and also to any software tools used in the specification, development, integration, test and maintenance of the system.

There is a metric, called the Maturity Index [4], which gives an indication of the degree of functional change incorporated into a version of a software system. This measure is used to quantify the readiness of software. Changes from a previous baseline to the current baseline are an indication of the current product's stability.

$$\text{maturity index} = \frac{M_T - (F_a + F_c + F_d)}{M_T}$$

Where M_T = number of software functions in current delivery

F_a = number of additional modules

F_c = number of changed modules

F_d = number of deleted modules

There are several other qualitative measures which can be applied to existing software. Some examples of these are listed below:

i) the age of the current version of the software in use;

ii) the average time between version releases;

iii) the average number of faults found and reported between version releases; and

iv) the overall failure history.

4. Collection of Data

Clearly, any attempt to predict and measure cost and productivity in order to gain control over the software process, will always be dependent upon careful data collection. Recently, this has been recognised as a discipline in itself and a number of experts in the field have published books on the subject, e.g. Grady and Caswell [2], Basili and Weiss [16], and Mellor [17].

Collection of data is dependent on the type of project, the project lifecycle and the quality and skills of the personnel involved. Depending upon the number and type of metrics and the particular circumstances of the project there are both manual and automated methods of collection. Both methods are discussed below.

4.1 Manual Collection Methods

Manual methods of collecting data are generally laborious and involve completing data forms with metric information. The stage at which such information is obtained with respect to the product lifecycle should be defined. It is important that metric collection procedures are developed and adhered to in order to give confidence in the final results.

If procedures are not followed correctly there is a danger of inconsistencies occuring which may distort the overall results. Methods of checking that procedures are followed also need to be developed otherwise common working practice may diverge from the stated procedures. If such divergence occurs inaccuracies can easily be introduced. If results are not properly documented due to inaccuracies, either in actual values or in the methods of collection, then the benefit of the exercise may be reduced or the results may actually be harmful to the project development due to inaccurate feedback. If inaccurate feedback of information occurs regularly then project confidence may deteriorate.

When procedures are not followed inconsistencies can occur simply as a result of a change of staff, due to slightly different working practices. Inaccuracies may occur from a member of staff being unaware of procedures or not having enough in-depth knowledge of the project to be able to compile metric results clearly.

Data forms must be specifically tailored to the metric information and be designed for ease of use by the user. The following points need to be considered when producing such forms:

i) administration - the relevant project information must be available;

ii) data collection - this task must be quick and easy;

iii) data analysis - how different metrics and forms relate to each other should be clear. Data reduction and presentation techniques could be incorporated; and

iv) notes - there should be adequate space for notes to be written.

Above all, data forms should be tailored to make it easier for the analyst to retrieve and analyse the information in a meaningful way.

Once data forms have been collated they must be filed/registered in an orderly fashion with respect to relevant configuration/project management rules.

For manually collected data to be easily used it is usually better to transfer such information to a database. Software tools can then be used to display and aid interpretation of the primary data. Such interpretation is discussed in section 5.

The choice of database on which to load the primary data depends both upon the project and the company involved. The database should be able to interact with the existing tools used within the company and have relational properties to enable data to be linked effectively.

4.2 Automated Collection Methods

Automation of metric data collection reduces the labour involved and therefore usually increases accuracy due to a decrease in human error and inconsistencies. Automation may also increase the overall coverage of results and the time taken to compile results may be shortened. If software development tools are already in use on a project they can sometimes be used as part of the data collection process. For example, static and dynamic analysis tools can also be used to collect data on specific metrics and such information fed into the metric results.

There are several data collection tools commercially available. A major advantage of using such tools is their ability not only to store the data, but to display the results in several formats thus allowing the data to be interpreted more easily (see section 5). The use of process related CASE tools has the added advantage that the metrics are automatically gathered as a part of the software development process. This includes both process and product metrics.

4.3 Verification of Data

It is important that the collected data is correct in order that the results produced can be given a degree of confidence. It is therefore beneficial to have an independent method of checking the integrity of the metric data as it is being produced.

Manually collected data is prone to errors. Procedures should be in place to reduce these errors by checking the completed forms and the final acceptance of results should be provisional on some form of checking. After the data has been input to a database it can be checked using software tools to display the information graphically. Human error in collecting data manually shows up easily in graphs as spurious results. Simple graphs that a spreadsheet or database package can produce are usually sufficient for this purpose.

Spreadsheets and databases are excellent for verification of manually collected data. Graphical representations of data can identify repeated patterns and anomalous values, and can convey results clearly and quickly to data collectors, project managers and third party assessors, for discussion.

5. Interpretation of Data

There are three stages in the process of interpreting collected data:

i) presentation;

ii) assessment; and

iii) validation.

These stages are discussed in the following subsections.

5.1 Presentation

Effective presentation of results is essential for reliable and easy interpretation. If results are presented badly they may be confusing and thus get misinterpreted. If results are consistently presented badly the overall impression of metrics within the workforce may deteriorate to such an extent that the results are not trusted.

If collected metric data can be analysed in subsets then this will aid clarity. However, it is important not to split related data. For a particular chosen subset all collected data must be shown in order to view trends and to make sure outlying values are included. It is the outlying values which are sometimes important in identifying mistakes.

Graphical methods of presentation are the most user friendly. There are a variety of graph types one can use such as line graphs, histograms, scatterplots and pie charts. Automated tools may use any of these or produce types of graph found only within the tool which are specific to the metric data being used.

The type of graph chosen often aids interpretation greatly. For example pie charts are excellent at portraying percentages whereas histograms can portray exact numbers easily and line graphs show trends well.

Knowledge of the way the metrics were collected and knowledge of the overall project is important when interpreting graphs. A histogram may show that a particular module has a higher error rate than other modules, it may be that this particular module is tested more often than others. A better representation would be the error rate as a percentage of the number of tests. Unless the background of how graphs were obtained is known results may be misinterpreted.

Once the data has been presented in the correct manner and sufficient background information is known as to why these particular metrics were chosen then the results can be assessed (see section 5.2).

The results should be presented in such a way as to clearly show up any metric values which lie outside the predefined limits. Automatic tools tend towards this type of presentation, but if the metrics have been collected manually and are being presented using general purpose packages, then care must be taken so that the method of presentation shows such outside values clearly.

5.2 Assessment

Once the collected data has been presented in a suitable manner it can be assessed. This process should be aided greatly by the type of presentation.

All metric results should be displayed to aid assessment. Values outside the predefined limits should be noted and the circumstances of such values examined in order to determine why they are outside the limits. There may be a valid reason for an outside value in which case the metric plan may need alteration, for example to change the limits. Validation is discussed in section 5.3.

If a value is outside the set limits, but the responsible engineer decides that it is acceptable, then a justification must be given, and documented, as to why the outlying value is acceptable in the particular circumstances present. A justification should always be given whether or not the metric plan is changed.

If there is always a justification for a certain metric value to be 'out' for particular circumstances then good documentation will show this up and may lead to alteration of the metric plan.

It is important for assessment to be performed on all collected values because the data may show undesirable trends. A set of values may be within the set limits but show a trend of approaching a limit. The assessment engineer may then decide, on experience, to investigate further and perhaps create new metrics to show up any effects more clearly.

This treatment of metrics falls into the realm of statistical process control. Caplen [18] gives an introduction to the subject and provides guidance on choosing suitable control limits and on sampling, plotting and analysing data.

There is an underlying assumption that data collected from a mature process follows a normal distribution. Taking action to eradicate any factor which tends to produce a non-normal distribution (for example outliers and skews) will tend to improve the overall quality.

5.3 Validation

In this context, validation is of the metrics themselves rather than the results. After collected data has been presented and assessed the metrics used should be validated to some extent. This is simply a check as to whether the metrics chosen are suitable for the task on which they are being used.

Collected data can be compared with what was originally expected or with an expected correlation to other data. Alternatively, engineers involved in the project can give a subjective opinion as to whether the metric corresponds to the objectives in mind.

This validation process is fairly loose and should be repeated as more data is collected and results are fed back into the process.

5.4 Example of Interpreting Data

The results of metrics, particularly code metrics which lay outside the upper and lower limits assigned, should be examined to investigate the reasons why.

For example, consider components A and B which both possess a cyclomatic complexity, V(G), of 36, which is thrice the upper limit of 12. On initial viewing of the results, an uninformed recommendation would be to rewrite both.

However, on examination, it is found that component A contains numerous jumps and exit points whereas component B contains a single CASE statement with 35 branches. Each branch in component B contains a single statement calling another component. The approach used in component A is complicated whilst that for component B is simple, readable and maintainable. Hence the metric value for component B is acceptable whereas the value for component A is not.

6. Utilisation of Data

There are two steps for the successful utilisation of collected data:

i) relating data to original objectives; and

ii) feedback of data into the development process.

Both of these stages are discussed below in the following sections.

6.1 Relating Data to Original Objectives

This report only deals with methods of collection, interpretation and feedback of metric data. However, it should be stressed here that there should be an underlying metric plan which states explicitly which metrics are to be used, where they are to be applied, and the expectations the project engineers have as to their use. These objectives should be fully documented.

Metrics are often used which define complexity of software. The underlying objectives of these metrics is to put a figure on the level of complexity of a piece of software. It is generally believed that complexity is inversely related to reliability because more complex software has a greater probability of faults remaining after release. Such metrics as "number of lines of code per module" and "number of modules" are used as a measure of complexity. Such metrics can easily be related to the original objectives in a vague manner, but to explicitly relate them, the objectives must be fully documented. There must be "levels of judgment" along with each objective in order for a metric to be useful.

For example, knowing the number of lines of code per module allows an engineer to judge complexity, but for this to be useful there must be a limit above which it is judged the module is too complex and should therefore be reassessed.

Such judgments should not be hard rules. Metrics simply allow an engineer to pinpoint possible trouble areas. Such areas should be assessed on an individual basis as to whether action is necessary. The comparison of metric results to original objectives allows such analysis, but only if the objectives are detailed enough.

If metrics are applied without objectives it confuses the analysis process and may ultimately have a detrimental effect on the metric plan.

Difficulty arises when individual metric results are combined in order to show trends. Such cross analysis must be fully documented with respect to the methods and reasons in order for results to be traceable and useful.

6.2 Feedback of Data

Feedback of information derived from metric analysis is a major part of analysing a system. Without feedback the information gained is of no use.

Figure 1 shows a basic feedback diagram. Both process and product metrics use the same feedback principles. The collection of information is performed via metric data collection either by manual or automated means, see section 4. The analysis of such information is performed as discussed in section 5.

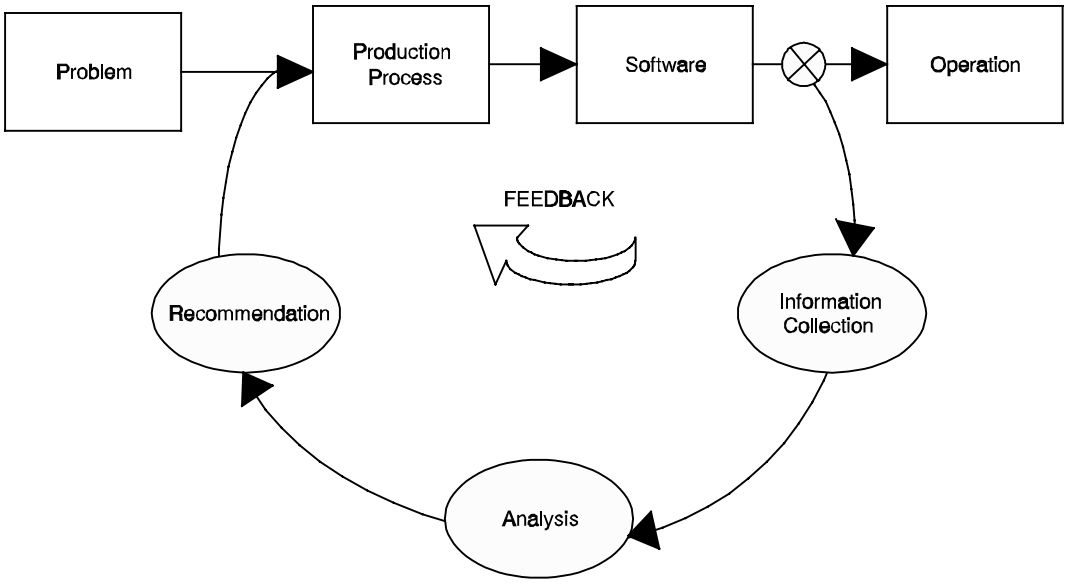

FIGURE 1 Basic Feedback Diagram

An important part of information feedback is composing the analysed data into recommendations which can then be fed back into the production process.

If the metrics have been properly chosen, and upper and lower limits set, then interpretation is relatively straight forward. However, it is rarely the person who collects the data who interprets it. Raw data is assessed and then presented in the form of reports. It is the format of these feedback reports which is critical. They must convey all relevant information in an easy to follow style. It is here that the presentation of data, discussed in section 5.1, is important.

The style of reports should be governed by the types of metrics being used. A combination of graphs with relevant text explaining the major points which the data represents is far more informative than a set of raw figures.

Such reports should be issued whenever metric data is available. The collection of metric data and hence timing of reports should be stated in the metric plan and linked to the project lifecycle.

The project plan or metric plan should determine the following aspects of metric analysis:

i) who collects the data;

ii) who analyses the data;

iii) who writes the reports;

iv) the number and type of meetings to be held to discuss the data;

v) who should be present at the meetings;

vi) the output from meetings;

vii) the format of any documents, with the responsible person identified; and

viii) the timing of meetings and documents.

It is not necessary to name specific people for all aspects of a project, but the level of authority must be designated for all stages. However it is normal practice to assign personnel names to high level process roles, usually in the quality plan.

One of the most important aspects is who has the ultimate responsibility of defining the results of the metric analysis. For all data presented there must be a responsible engineer who decides the appropriate course of action. BS5750 [19] and TickIT [20] emphasise qualifications and vocational training to give confidence that personnel are capable of performing their relevant tasks. Such training is highly advantageous when ensuring confidence both within the project and externally.

Action can range from re-writing specific areas of code so they come within the metric limits to doing nothing because all limits have been met or an acceptable justification exists. It is important for all decisions to be recorded for ease of traceability.

The objective of utilising metrics is to determine whether or not the original goals are being met. If objectives are not being met then the reasons why need to be understood, recorded and disseminated. Even if objectives are being met the addition of further metrics to the metric plan can be used as a second stage to analysis.

The addition of new metrics is very useful once more detailed knowledge of the situation has been gained. Metrics should be validated, as discussed in section 5.3, to check if they are the correct ones for the task. The process of checking existing metrics may lead to the additions of new ones.

The addition of any further metrics should be fully documented with reasons and the metric process repeated.

6.2.1 Trend Analysis

An important aspect of metrics is that they can be used to analyse project trends very effectively. This is particularly useful for process metrics whereby the process can be regulated over time by assessing metric results against present timescales and milestones. If the results of a certain metric are viewed over time in conjunction with its boundary conditions then corrections may be possible before the end of the production process. This allows production to be brought back on course and so avoid costly errors and major re-work.

Trend analysis is made easier and more effective with correct use of presentation techniques, such as graphs and flow charts; as discussed in section 5.1.

7. References

[1] *Software Metrics a Rigorous Approach*, N.E. Fenton, Chapman & Hall, 1991.

[2] *Software Metrics: Establishing a Company-wide Program*, Grady RB, Caswell DL, Prentice Hall, NJ, 1987

[3] *Application of Metrics in Industry: Metric Users Handbook*, Centre for Systems and Software Engineering, South Bank Polytechnic, London, 1992.

[4] IEEE, 982.1-1988, *Standard Dictionary of Measures to Produce Reliable Software*

[5] ISO/IEC DIS 9126, Draft International standard, *Information Technology - Software Product Evaluation - Quality Characteristics and Guidelines for their Use*

[6] *Characteristics of Software Quality*, B W Boehm, TRW North Holland, 1975.

[7] *Factors in Software Quality*, J A McCall, General Electric, no 77C1502, June 1977.

[8] *Software Engineering Economics*, Boehm BW, Prentice Hall, New York, 1981

[9] *Elements of Software Science*, M H Halstead, North Holland, Elsevier 1977.

[10] "A Complexity Measure", T J McCabe, IEEE Transactions on Software Engineering, Volume 1 SE-2 pp 308-320, December 1976.

[11] BS5760 Part 8, (Currently published as DD198), *Assessment of Reliability of System Containing Software*

[12] *Software Reliability: Principles and Practices*, Glenford J Myers, Wiley & Sons, 1976.

[13] *Software Engineering*, Martin L Shooman, McGraw-Hill, 1983.

[14] "Software Structure Metrics Based on Information Flow", S Henry and D Kafura, IEEE Trans on Software Engineering, Volume SE-7, 1981.

[15] "Flow Diagrams, Turing Machines and Languages with Only Two Formulation Rules", C Bohm and G Jacopini, Communications of the ACM Volume 9, 1966.

[16] "A Methodology for Collecting Valid Software Engineering Data", Basili VR, Weiss DM, IEEE Trans Soft Eng SE-10(6), 1984, 728-738

[17] "Software Reliability Data Collection: Problems and Standards", Mellor P, in Pergammon Infotech State-of-the-art Series 'Software Reliability', 1985, 165-181 and 256-257

[18] *A Practical Approach to Quality Control*, R H Caplen, Century Business, Fifth Edition, ISBN 0 09 173581 5

[19] BS5750 *Quality Systems*, British Standards Institution, 1987

[20] *TickIT, Guide to Software Quality Management System Construction and Certification using ISO 9001/EN29001/BS5750 Part 1 (1989)*, Issue 2.0, February 1992

[21] "Information Flow Metrics for the Evaluation of Operating Systems Structure", S Henry, PhD Theses, Department of Computer Science, Iowa State University, Ames, 1979.

[22] *Software Engineering: Metrics and Models*, S D Conte et al, Benjamin/Cummins Publishing Company, 1986.

[23] MOD, Interim Defence Standard 00-55, *The Procurement of Safety Critical Software in Defence Equipment*, Part 1:Requirements, Issue 1, 1991.

[24] MOD, Interim Defence Standard 00-55, *The Procurement of Safety Critical Software in Defence Equipment*, Part 2:Guidance, Issue 1, 1991.

8. Glossary

CSC	Component Stress Complexity
DDP	Decision to Decision Path
IB	Instruction Block
IEEE	The Institute of Electrical and Electronic Engineers
LCSAJ	Linear Code Sequence and Jump
MISRA	Motor Industry Software Reliability Association
PPP	Procedure to Procedure Path
ROM	Read Only Memory

/	Division
*****	Multiplication
n	Vocabulary Size
$\mathbf{n_1}$	Number of Distinct Operators
$\mathbf{n_2}$	Number of Distinct Operands
N	Component Length
$\mathbf{N_1}$	Number of Operator Occurrences
$\mathbf{N_2}$	Number of Operand Occurrences
V(G)	Cyclomatic Complexity

Acceptance Criteria	The thresholds or limits which the results of applying a metric must satisfy.
Attribute	A feature or property of the software or development process.
Complexity	The complexity of software provides an indication of the mental effort required to understand both the syntactic and semantic content of the software.
Component	The lowest constituent part of software or hardware which performs a function. For example, in software, a procedure, function or subroutine and in hardware, a diode, capacitor, etc.
DDP (Decision to Decision Path)	A set of statements whose origin is either the entry point to the software component or a decision taken by a control instruction in the software component and whose end is either the next decision or an exit from the software component.
Fault Tolerance	The extent to which the system and its components have an inbuilt capability to provide continued correct execution in the presence of a number of faults.
Instruction Block	A sequence of statements contained in a software component.
LCSAJ (Linear Code Sequence and Jump)	A sequence of statements followed by a jump in a software component.
Maintainability	The extent that a software system, in service, can be understood and changed in a controlled manner to allow the correction of errors or implementation of required enhancements.
Measure	The result of applying a metric.
Measurement	Quantitative means of obtaining an indication of a particular aspect or characteristic of a system.
Metric	A formula for obtaining a quantitative indication of a particular aspect or characteristic of a system.
Modularity	The extent to which a software system is comprised of self-contained components.
Module	A part of a system which is made up of one or more software or hardware components.

Operand A symbol representing data, for example a variable name.

Operator A symbol or keyword which specifies an algorithmic action.

PPP A calling relationship between two software components
(Procedure to Procedure Path)

Software Reliability The probability that during a given period of time the software systems, subsystems and components perform their required functions under given conditions.

Structuredness The extent to which a software system or subsystem is decomposed into a hierarchy of levels of abstraction and detail.

Testability The extent to which the software system, subsystem or component can be tested for compliance against its requirements.

Thresholds Upper and/or lower bounds for metrics.

Understandability The extent to which the software source code and its documentation can be comprehended by a competent software engineer other than its author.

Appendix A - Selected Product Metrics

Contents

A.1 Introduction

This Appendix describes a wide range of product metrics. Each metric is defined in terms of the following subsections.

1. **Aim**	a statement of the aim of the metric and the attributes to which it may contribute;
2. **Description**	a brief précis of the history and use of the metric and a description of the metric which includes, where appropriate, the formulae for its use;
3. **Acceptance Criteria**	the thresholds which are acceptable or those recommended for the metric results to meet. Where evidence is lacking and no judgment can be found a recommendation is provided for the acceptance criteria; and
4. **Restrictions**	a description of any problems in the use of the metric or any common reason for the non-achievement of the acceptance criteria.

Table 1 cross references the identified metrics to the software attributes.

Table 2 details whether the metric can be applied to a software component, a software system or subsystem or both.

Table 3 specifies the acceptance criteria for the metrics which are appropriate for indicating whether the relevant attribute has been satisfied. Where the metric can contribute to more than one attribute the same acceptance criteria will be used.

If a software component does not meet the acceptance criteria then a justification should be made in the form of an explanation of the reasons for non-compliance. If the justification is not acceptable then the component may require rework or retest until it achieves its acceptance criteria.

An overall qualitative judgment should be made for all of the software attributes to identify whether they have been satisfied. This judgment should be based on the results of applying the methods and metrics prescribed for each attribute of the software in Section 3 and detailed on Table 2.

Notes on Metrics

Note 1 - Control Flow Graph

The control flow graph is a directed graph representing the logical structure of a component. The node symbols represent different statements or sequence of statements and the edges represent the transfer of control between these entities.

Note 2 - System Call Graph

A call graph is a directed graph where each node represents a component of the program and a directed edge between two nodes represents a call from a calling component to a called component. The graph is represented as a tree structure and calls are made from higher levels to lower levels.

Note 3 - Halstead Software Science Metrics

The confidence given to the values of the various software science metrics [9] depends upon the purity of the algorithmic implementation. Halstead [9] identified the following impurities;

i) complementary operators: one operator cancels the effect of another operator and the variable used in the operation has not been used since the last operation;

ii) ambiguous operands: the same operand represents different variables in the algorithm;

iii) synonymous operands: the same variable has different operand names;

iv) unrepresented expression: a repeated expression has no corresponding variable assignment;

v) unnecessary assignment: the calculated result of an expression is only used once but is assigned to a variable; and

vi) unfactorised expressions: there is no factorisation in a complex expression.

A.2 Product Metrics Definitions

A.2.1 Average Statement Size

A.2.1.1 Aim

To obtain an indication of the complexity and understandability of the software by examining the size of each statement line.

A.2.1.2 Description

No usage information is available at present for this metric.

The metric indicates the average size of statements in a software component and is obtained by dividing Halstead's component length metric by the number of statements.

A.2.1.3 Acceptance Criteria

Experience gained indicates that the average statement size should be less than 9 operands and operators per statement. If an average is obtained which is greater than 8, this implies either that the majority of statements are complex or several statements are very complex.

A.2.1.4 Restrictions

This metric may be difficult to apply to an assembler language due to its inherent nature. The format of assembler code is consistent in that it can possess a label, an operator and one or two operands.

A.2.2 Comment Frequency

A.2.2.1 Aim

To provide an indication of the understandability of software components by measuring the extent to which the software is adequately commented.

A.2.2.2 Description

Good engineering practice dictates that all software is commented. Each component should possess a component header which describe the component's function, interfaces and revisions. In addition an adequate level of in-code comments should be maintained.

The Comment Frequency is the ratio of in-code comments to statements in a software component, that is:

$$\frac{\text{number of comments}}{\text{number of statements}}$$

This ratio does not therefore include the component header and declaration comments since if these are included a true metric result for the in-code comments will not be obtained.

A.2.2.3 Acceptance Criteria

Experience gained indicates that the comment frequency, for a high level language, should be at least 1 comment every 2 lines of code. For assembler languages the criteria should be increased to 1 comment per line of code.

A.2.2.4 Restrictions

The metric only measures the frequency of the comments. It cannot ensure that comments are accurate or helpful. This has to be examined as part of a code review.

A.2.3 Component Length (N)

A.2.3.1 Aim

To aid the evaluation of a component's understandability and textual complexity in terms of its operands and operators.

A.2.3.2 Description

The component length metric is one of Halstead's [9] software science metrics which is derived from two of Halstead's basic counts, that is operators and operands.

The length of the component expressed in operators and operands is;

$$N = N_1 + N_2$$

Where N_1 is the number of operator occurrences and N_2 is the number of operand occurrences.

A.2.3.3 Acceptance Criteria

The acceptance criteria for the length of a component has been set after consideration of the acceptance criteria for the operand and operator occurrence metrics. This indicates that each component's length should be between 3 and 250.

A.2.3.4 Restrictions

See Section 1. Note 3 - Halstead Software Science Metrics.

A.2.4 Component Stress Complexity

A.2.4.1 Aim

To aid in evaluating the complexity of a component in terms of its internal structure and its external connections.

A.2.4.2 Description

Henry and Kafura [14] defined a metric which depends on the complexity of pseudo code and the complexity of the component's connections. They used, initially, a simple program length measure for the component complexity, although other software complexity measures can be used such as Halstead's component length or McCabe's V(G), to improve accuracy. A component's connection with the system was expressed as a measure of the fan-in and fan-out of the component. The fan-in is the number of components which call the component and the fan-out is the number of components called by the component.

This metric was used by Henry and Kafura on version 6 of the Unix Operating System [21].

The Component Stress Complexity (CSC) has been adapted from the initial work performed by Henry and Kafura and currently uses McCabe's complexity metric.

The CSC is calculated by:

$$CSC = \text{Cyclomatic Number} * (\text{Fan-In} * \text{Fan-Out})^2$$

In the calculation the Fan-In and Fan-Out figures must have a minimum value of 1.

A.2.4.3 Acceptance Criteria

No acceptance criteria have been set for this metric as it aims to provide an indication of potential stress components in a software system. Experience suggests that any component which possesses a CSC greater than 10 000 should be investigated.

A.2.4.4 Restrictions

Library routines may have a high stress complexity if they call other library routines as they may have a high fan-in.

A.2.5 Cyclomatic Number V(G)

A.2.5.1 Aim

To aid in evaluating the complexity, structuredness and testability of a software component based on the component's control flow.

A.2.5.2 Description

McCabe developed a mathematical technique [10] to provide a quantitative basis for the evaluation of software components and to identify components where test and maintenance would be difficult.

McCabe's cyclomatic complexity metric is based on the number of paths through a component and is defined using graph theory in terms of basic paths which can generate every possible path of a component.

For a strongly connected control flow graph (see Section 1 Note 1 - Control Flow Graph), that is a control flow graph with one entry and one exit point;

$$V(G) = \text{Number of Edges} - \text{Number of Nodes} + 2$$

Several properties of the cyclomatic complexity are listed below;

i) V(G) >= 1;

ii) V(G) is the maximum number of linearly independent paths in G; it is the size of the base set;

iii) Inserting or deleting functional statements in G will not affect V(G);

iv) G has only one path if and only if V(G)=1;

v) Inserting a new edge in G increases V(G) by 1; and

vi) V(G) depends on the decision structure of G.

The main advantage of McCabe's complexity measure is that it is a simple concept and easy to calculate.

A.2.5.3 Acceptance Criteria

The cyclomatic complexity has been used previously to help limit the complexity of components in software projects. Experience and empirical data suggest that there is a step function in the density of errors above a cyclomatic complexity of 10 [22]. However, limiting the cyclomatic complexity to 10 impacts the number of components in the software system. The individual components are easy to maintain but the calling structure is less understandable. Therefore the acceptance criteria proposed is a maximum cyclomatic complexity of 15.

A.2.5.4 Restrictions

The use of a CASE (or similar, for example SWITCH in the 'C' programming language) statement in a high level language impacts upon the value of the cyclomatic complexity. The cyclomatic complexity is increased by the number of branches in the CASE statement.

One method of avoiding this is by reducing the cyclomatic complexity by the number of branches in the CASE statement and adding 1 for the CASE statement. This modified cyclomatic complexity takes into account the CASE statement and any inherent complexity in its branches but ignores the number of branches.

The CASE statement is preferable to multiple IF-THEN-ELSE statements. Consider two examples. The first example contains three nested IF-THEN-ELSE statements and the second contains a CASE statement with 3 branches. Both of these examples possess a V(G) of 4. However, if the new CASE rule proposed is used then the V(G) of the second example becomes 2. The only restriction is that this figure does not now represent the minimum number of test cases required to test the component.

A.2.6 DDP Coverage

A.2.6.1 Aim

To show a high degree of testing and thus give confidence in the testing performed.

A.2.6.2 Description

The DDP Coverage is the percentage of decision to decision paths which have been executed. This can be calculated for the individual software component, the software subsystem or the system as a whole.

A.2.6.3 Acceptance Criteria

Experience gained indicates that each component should achieve a DDP coverage of at least 80%.

A.2.6.4 Restrictions

It may be impossible to test all decision to decision paths in a software system due to defensive programming techniques. Two examples are given of defensive programming situations where achieving full testing coverage would be difficult:

i) It may be standard practice to insert trace statements into the code which are only executed when the trace flag is on. These are used at unit testing to output a message to produce a record of the execution sequence. These trace statements are implemented as IF-THEN statements, thus due to the trace flag only half of the decision can be executed; and

ii) Code may be produced to cope with out of the ordinary events such as an operating system or Read Only Memory (ROM) becoming corrupted. The condition required to enable the code to be executed has to be set up, that is corrupting an operating system or ROM whilst the system is running. In practice this is extremely difficult if not impossible to achieve.

A.2.7 Essential Cyclomatic Complexity

A.2.7.1 Aim

To ensure that a component is reducible and thus has been developed using structured programming.

A.2.7.2 Description

The essential cyclomatic complexity is obtained in the same way as the cyclomatic complexity but it is based on a 'reduced' control flow graph.

The purpose of reducing a graph is to check that the component complies with the rules of structured programming.

A control graph which can be reduced to a graph whose cyclomatic complexity is 1 is said to be structured. Otherwise reduction will show elements of the control graph which do not comply with the rules of structured programming.

The principle of control graph reduction is to simplify the most deeply nested control subgraphs into a single reduced subgraph. A subgraph is a sequence of nodes on the control flow graph which has only one entry and exit point. Four cases are identified by McCabe [10] which result in an unstructured control graph. These are;

i) a branch into a decision structure;

ii) a branch from inside a decision structure;

iii) a branch into a loop structure; and

iv) a branch from inside a loop structure.

However, if a component possesses multiple entry or exit points then it can not be reduced. The use of multiple entry and exit points breaks the most fundamental rule of structured programming.

A.2.7.3 Acceptance Criteria

The essential cyclomatic complexity for a component should be 1.

A.2.7.4 Restrictions

The essential cyclomatic complexity only applies to high level languages. It cannot be applied to assembler languages. Due to the nature of assembler languages it is difficult not to use unconditional jumps or multiple exit points.

A.2.8 Hierarchical Complexity

A.2.8.1 Aim

To aid in the evaluation of the complexity of a call graph.

A.2.8.2 Description

This metric applies to the system call graph (see Section 1 Note 2 - System Call Graph).

The Hierarchical Complexity is the average number of software components per calling level, that is:

$$\frac{\text{number of software components}}{\text{number of calling levels}}$$

A.2.8.3 Acceptance Criteria

Experience gained indicates that the Hierarchical Complexity should be less than 5, that is, between 1 and 5 components per calling level.

A.2.8.4 Restriction

None known.

A.2.9 IB Coverage

A.2.9.1 Aim

To show a high degree of testing and thus give confidence in the testing performed.

A.2.9.2 Description

The IB Coverage is the percentage of instruction blocks which have been executed. It can be calculated for the individual software components, the software subsystem or the system as a whole.

A.2.9.3 Acceptance Criteria

Experience gained indicates that each software component should achieve an IB coverage of 100%, that is all statements in the software component should be executed at least once.

A.2.9.4 Restrictions

It may be impossible to test all instruction blocks in a software system due to defensive programming techniques. Two examples are given of defensive programming situations where achieving full testing coverage would be difficult.

i) It may be standard practice to insert trace statements into the code which are only executed when the trace flag is on. These are used at unit testing to output a message to give an indication of where the current execution is currently at. These trace statements are implemented as IF-THEN statements, thus due to the trace flag only half of the decision can be executed.

ii) Code may be produced to cope with out of the ordinary events such as an operating system or ROM becoming corrupted. The condition required to enable the code to be executed has to be set up, that is corrupting an operating system or ROM whilst the system is running. In practice this is extremely difficult if not impossible to achieve.

A.2.10 LCSAJ Coverage

A.2.10.1 Aim

To show a high degree of testing and thus give confidence in the testing performed.

A.2.10.2 Description

The LCSAJ Coverage is the percentage of Linear Code Sequences and Jumps which have been executed. It can be calculated for the individual software components, the software subsystem or the system as a whole.

A.2.10.3 Acceptance Criteria

Experience gained indicates that each software component should achieve a LCSAJ coverage of at least 60%.

A.2.10.4 Restrictions

It may be impossible to test all LCSAJs in a software system due to defensive programming techniques, in the same manner as DDP coverage.

A.2.11 Number of Calling Levels

A.2.11.1 Aim

To identify and minimise the level of calling complexity on a system call graph (see section 1 Note 2 - System Call Graph) by limiting the number of calling levels allowed.

A.2.11.2 Description

This metric is the number of calling levels which exist on a call graph for a software system or subsystem.

A.2.11.3 Acceptance Criteria

Experience gained indicates that the total number of calling levels on a system call graph should not be greater than 8.

A.2.11.4 Restrictions

None known.

A.2.12 Number of Calling Paths

A.2.12.1 Aim

To identify the number of calling paths on a system call graph (see section 1 Note 2 - System Call Graph).

This will give an indication of the testing effort required in order to execute each calling path and hence obtain a Procedure to Procedure coverage of 100%.

A.2.12.2 Description

This metric is the number of calling paths which exist for a software system or subsystem.

A.2.12.3 Acceptance Criteria

Experience gained indicates that the total number of calling paths in a system or sub-system should not be greater than 250. If more calling paths exist then extensive testing is required.

A.2.12.4 Restrictions

There are restrictions on the application of this metric to large systems but this is not a consideration within the automotive industry since the systems installed on vehicles tend to be small.

A.2.13 Number of Components

A.2.13.1 Aim

To help evaluate complexity, testability and understandability. The more components a software system possesses the more complex it is to understand and test.

A.2.13.2 Description

This metric is the number of software components which are contained in software system or subsystem.

A.2.13.3 Acceptance Criteria

Experience gained indicates that the total number of components in a software system or subsystem should not be greater than 150. If more components exist then more extensive testing is required.

A.2.13.4 Restrictions

None known.

A.2.14 Number of Decision Statements

A.2.14.1 Aim

To help evaluate the complexity of the component and estimate the amount of testing necessary.

A.2.14.2 Description

This metric is the number of decisions statements in a software component. A decision statement could be a IF-THEN, WHILE, REPEAT, CASE etc.

A.2.14.3 Acceptance Criteria

Experience gained indicates that the total number of decision statements in a software component should be less than 9.

A.2.14.4 Restrictions

None known.

A.2.15 Number of Distinct Operands (n_2)

A.2.15.1 Aim

To help evaluate the complexity, testability and understandability of the component by examining the number of unique operands necessary for the software component to perform its required function.

A.2.15.2 Description

The number of distinct operands is one of the basic counts upon which Halstead's software science metrics are based [9]. Halstead considered that a component is a series of tokens which can be defined as either operators or operands.

This metric is the number of distinct operands used in a software component where an operand can be any symbol representing data.

A.2.15.3 Acceptance Criteria

After analysis of a number of sources, it would appear that a sensible limit for the number of distinct operands in a component is 50. If this limit is broken then the understandability of the component will be reduced.

A.2.15.4 Restrictions

See Section A.1 Note 3 - Halstead Software Science Metrics.

A.2.16 Number of Distinct Operators (n_1)

A.2.16.1 Aim

To help evaluate the complexity and understandability of the component by examining the number of unique operators necessary for the software component to perform its required function.

A.2.16.2 Description

The number of distinct operators is one of the basic counts upon which Halstead's software science metrics are based [9]. Halstead considered that a component is a series of tokens which can be defined as either operators or operands.

This metric is the number of distinct operators used in a software component where an operator can be any symbol or keyword that specifies an algorithmic action.

A.2.16.3 Acceptance Criteria

After analysis of a number of sources, it would appear that a sensible limit for the number of distinct operators in a component is 35. If this limit is broken then the understandability of the component will be reduced.

A.2.16.4 Restrictions

See Section A.1 Note 3 - Halstead Software Science Metrics.

A.2.17 Number of Entry Points

A.2.17.1 Aim

To help ensure that the component is structured.

A.2.17.2 Description

Good programming practice requires that if a component is to be considered as structured then it shall only contain one entry point [4], [12].

This metric is a count of the number of entry points in a software component.

A.2.17.3 Acceptance Criteria

Structured programming requires that this value is 1.

A.2.17.4 Restrictions

None known.

A.2.18 Number of Execution Paths

A.2.18.1 Aim

To obtain an estimate of the testability of a software component and to help evaluate the structuredness of the software component.

A.2.18.2 Description

This metric indicates the number of non-cyclic execution paths in a software component (NPATH). It is calculated using the control transfers initiated by different types of statements.

NPATH for a sequence of statements at the same nesting level is the product of the NPATH values for each statement and for the nested structures, it is the sum of;

i) NPATH(*if then else*) = NPATH(body of *then*) + NPATH(body of *else*);

ii) NPATH(*while do*) = NPATH(body of *while*) + 1;

iii) NPATH(*case of*) = $\Sigma_{i=1,n}$ NPATH(body of *it case*)
where n is the number of cases of the case statement.

iv) NPATH(*sequence*) = 1.

The number of execution paths gives a more accurate idea than McCabe's cyclomatic number of the number of test cases required to fully test a component.

A.2.18.3 Acceptance Criteria

Experience gained indicates that the number of execution paths in a software component should be less than 75. If the number of execution paths is greater than this then it quickly becomes infeasible to fully test the component.

A.2.18.4 Restrictions

Although the number of execution paths gives an accurate number of the number of test cases required, it does possess certain limitations;

i) it cannot be calculated if there are any unconditional branches (GOTO's) in the component as this adds an infinite number of paths due to allowing loop backs; and

ii) it takes into account theoretical paths that cannot be executed in reality, for example IF TRUE THEN ... ELSE ..., where the ELSE branch should never be executed.

A.2.19 Number of Exit Points

A.2.19.1 Aim

To help ensure that the component is structured.

A.2.19.2 Description

Good programming practice states that if a component is to be considered as structured then it shall only contain one exit point [4], [12].

This metric is a count of the number of exit points in a software component.

A.2.19.3 Acceptance Criteria

Structured programming requires that this value is 1.

A.2.19.4 Restrictions

None known.

A.2.20 Number of Operand Occurrences (N_2)

A.2.20.1 Aim

To help evaluate the textual complexity of a software component.

A.2.20.2 Description

This metric is one of the basic counts upon which Halstead's Software Science metrics are based [9]. Halstead considered that a component is a series of tokens which can be defined as either operators or operands.

This metric is the number of occurrences of each operand in a software component.

A.2.20.3 Acceptance Criteria

Experience gained indicates that the total number of operand occurrences should be 120 or less. If this limit is exceeded then the component's understandability may become reduced.

A.2.20.4 Restrictions

See Section A.1 Note 3 - Halstead Software Science Metrics.

A.2.21 Number of Operator Occurrences (N_1)

A.2.21.1 Aim

To help evaluate the textual complexity of a software component.

A.2.21.2 Description

This metric is one of the basic counts upon which Halstead's Software Science metrics are based [9]. Halstead considered that a component is a series of tokens which can be defined as either operators or operands.

This metric is the number of occurrences of each operator in a software component.

A.2.21.3 Acceptance Criteria

Experience gained indicates that the total number of operator occurrences should be 140 or less. If this limit is exceeded then the component's understandability may become reduced.

A.2.21.4 Restrictions

See Section A.1 Note 3 - Halstead Software Science Metrics.

A.2.22 Number of Statements

A.2.22.1 Aim

To help evaluate the understandability of a component by examining the size of a software component.

A.2.22.2 Description

This metric is the number of statements in a software component.

A.2.22.3 Acceptance Criteria

Experience gained indicates that the number of statements in a software component should be 80 or less. If this limit is exceeded then the understandability of the component may be reduced.

A.2.22.4 Restrictions

The limit should be applied pragmatically since it may not always be suitable. For instance if a component possesses 100 statements and performs a single function, it is advisable that the component is not divided into two components.

A.2.23 Number of Structuring Levels

A.2.23.1 Aim

To help ensure that a software component is structured and to aid in measuring the complexity of the component.

A.2.23.2 Description

This metric is the number of levels of structuring in the software component and is determined from the use of control statements, such as IF, WHILE, FOR, CASE etc.

A.2.23.3 Acceptance Criteria

Experience gained indicates that the number of structuring levels in a software component should be less than 7.

A.2.23.4 Restrictions

None known.

A.2.24 Number of Unconditional Jumps

A.2.24.1 Aim

To ensure that a component is structured and to aid evaluating the effort required for testing the component. The greater the number of unconditional jumps the harder a component is to test and maintain.

A.2.24.2 Description

This metric is the number of times a software component uses a jump statement (GOTO) to alter its control flow, that is avoid execution of a set of statements.

Good programming practice states that unconditional jumps should not be used [12], [13] and some standards mandate this [23], [24].

A.2.24.3 Acceptance Criteria

No unconditional jumps should be used in a software component.

A.2.24.4 Restrictions

This acceptance criteria is readily applicable only to high level languages. Due to the nature of low level languages unconditional jumps may be required but should be avoided if at all possible.

The use of unconditional jumps in low level languages is not recommended and should be avoided if at all possible. Where it is unavoidable their usage should be fully justified in the documentation supporting the relevant software module and an increased comment frequency should be evident in the affected region of the code.

A.2.25 PPP Coverage

A.2.25.1 Aim

To show a high degree of testing and thus give confidence in the testing performed.

A.2.25.2 Description

The PPP Coverage is the percentage of Procedure to Procedure Paths which have been executed. This can be calculated for the software component, subsystem or the system.

A.2.25.3 Acceptance Criteria

A system PPP coverage of 100% is desirable, that is each procedure to procedure path should be called at least once.

A.2.25.4 Restrictions

It may be impossible to test all procedure to procedure paths in a software system due to defensive programming techniques. For instance, code may be produced to cope with out of the ordinary events such as an operating system or ROM becoming corrupted. The condition required to enable the code to be executed has to be set up, that is corrupting an operating system or ROM whilst the system is running. In practice this is extremely difficult if not impossible to achieve.

A.2.26 Structural Complexity

A.2.26.1 Aim

To help evaluate the complexity of a call graph.

A.2.26.2 Description

This metric applies to the system call graph (see Section A.1 Note 2 - System Call Graph).

The Structural Complexity is the average number of calls per software component; that is

$$\frac{\text{number of calls between software components}}{\text{number of software components}}$$

A.2.26.3 Acceptance Criteria

Experience gained indicates that the structural complexity should be between 0.5 and 3.

A.2.26.4 Restrictions

None known.

A.2.27 Vocabulary Size (n)

A.2.27.1 Aim

To help measure the textual complexity and the understandability of a software component in terms of operands and operators.

A.2.27.2 Description

This metric is derived from two of the basic counts upon which Halstead's Software Science metrics are based [9]. Halstead considered that a component is a series of tokens which can be defined as either operators or operands.

The vocabulary size metric is the total size of the vocabulary used, that is the number of distinct operators plus the number of distinct operands.

$$n = n_1 + n_2.$$

Where n_1 is the number of distinct operators and n_2 is the number of distinct operands.

A.2.27.3 Acceptance Criteria

The acceptance criteria for the vocabulary size of a component has been set after consideration of the acceptance criteria for the number of distinct operators and operands. On this basis each software component's vocabulary size should be between 3 and 75.

A.2.27.4 Restrictions

See Section A.1 Note 3 - Halstead Software Science Metrics.

TABLES

Software Attributes	Type of Technique	Area of Application	Technique or Metric
Complexity	Metrics	Component Source Code	Number of Statements Number of Distinct Operands Number of Distinct Operators Number of Operand Occurrence Number of Operator Occurrence Vocabulary Size Component Length Average Statement Size Cyclomatic Number Number of Decision Statements Number of Structuring Levels Component Stress Complexity
		System Source Code	Number of Components Number of Calling Levels Number of Calling Paths Hierarchical Complexity Structural Complexity
Maintainability	Method	System	Technical Audit
	Metrics	System and Component Source Code	Complexity Modularity Structuredness Testability Understandability
Modularity	Method	Component Source Code	Source Code Analysis
Reliability	Method	System	Design Audit Reliability Estimation

Continued . . .

TABLE 1

SOFTWARE ATTRIBUTES AND METHODS FOR THEIR ACHIEVEMENT

Software Attributes	Type of Technique	Area of Application	Technique or Metric
Structuredness	Method	Component Source Code	Interval Reduction
	Metrics	Component Source Code	Cyclomatic Number Essential Cyclomatic Complexity Number of Entry Points Number of Exit Points Number of Structuring Levels Number of Unconditional Jumps Number of Execution Paths
Testability	Metrics	Component Source Code	Cyclomatic Number Number of Distinct Operands Number of Unconditional Jumps Number of Execution Paths Number of Decision Statements IB Coverage DDP Coverage LCSAJ Coverage PPP Coverage
		System Source Code	Number of Calling Paths Number of Components IB Coverage DDP Coverage LCSAJ Coverage PPP Coverage
Understandability	Method	System Documentation	Documentation Audit
	Metrics	Component Source Code	Number of Statements Comment Frequency Number of Distinct Operands Number of Distinct Operators Vocabulary Size Average Statement Size Component Length
		System Source Code	Number of Components

TABLE 1 (cont)

SOFTWARE ATTRIBUTES AND METHODS FOR THEIR ACHIEVEMENT

Metric	Area of Application	Software Attributes
Average Statement Size	Component	Complexity, Understandability
Comment Frequency	Component	Understandability
Component Length	Component	Complexity, Understandability
Component Stress Complexity	Component	Complexity
Cyclomatic Number	Component	Complexity, Structuredness, Testability
DDP Coverage	Both	Testability
Essential Cyclomatic Complexity	Component	Structuredness
Hierarchical Complexity	System	Complexity
IB Coverage	Both	Testability
LCSAJ Coverage	Both	Testability
Number of Calling Levels	System	Complexity
Number of Calling Paths	System	Complexity, Testability
Number of Components	System	Complexity, Testability, Understandability
Number of Decision Statements	Component	Complexity, Testability
Number of Distinct Operands	Component	Complexity, Testability, Understandability
Number of Distinct Operators	Component	Complexity, Understandability
Number of Entry Points	Component	Structuredness
Number of Execution Paths	Component	Structuredness, Testability
Number of Exit Points	Component	Structuredness
Number of Statements	Component	Complexity, Understandability
Number of Structuring Levels	Component	Complexity, Structuredness
Number of Unconditional Jumps	Component	Structuredness, Testability
Number of Distinct Operands	Component	Complexity
Number of Distinct Operators	Component	Complexity
PPP Coverage	Both	Testability
Structural Complexity	System	Complexity
Vocabulary Size	Component	Complexity, Understandability

TABLE 2 PRODUCT METRICS AND THEIR APPLICATION

Software Metric	Area of Application	High Level Languages		Low Level Languages	
		Min	Max	Min	Max
Average Statement Size	Component	2	8	N/A	N/A
Comment Frequency	Component	0.5	1	1	1
Component Length	Component	3	250	3	250
Component Stress Complexity	Component	1	10000	1	10000
Cyclomatic Number	Component	1	15	1	15
DDP Coverage	Both	80%	100%	80%	100%
Essential Cyclomatic Complexity	Component	1	1	1	1
Hierarchical Complexity	System	1	5	1	5
IB Coverage	Both	100%	100%	100%	100%
LCSAJ Coverage	Both	60%	100%	60%	100%
Number of Calling Levels	System	1	8	1	8
Number of Calling Paths	System	1	250	1	250
Number of Components	System	1	150	1	150
Number of Decision Statements	Component	0	8	0	8
Number of Distinct Operands	Component	1	50	1	50
Number of Distinct Operators	Component	1	35	1	35
Number of Entry Points	Component	1	1	1	1
Number of Execution Paths	Component	1	75	1	75
Number of Exit Points	Component	1	1	1	1
Number of Operand Occurrences	Component	1	120	1	120
Number of Operator Occurrences	Component	1	140	1	140
Number of Statements	Component	1	80	1	80
Number of Structuring Levels	Component	1	6	1	6
Number of Unconditional Jumps	Component	0	0	0	0(2.24)
PPP Coverage	System	100%	100%	100%	100%
Structural Complexity	System	0.5	3	0.5	3
Vocabulary Size	Component	3	75	3	75

TABLE 3 - PRODUCT METRIC ACCEPTANCE CRITERIA